notes1: slides,

* File Permissions (35-38)
  + Discretionary Access Control (DAC)
    - Permissions for 3 Entities
      * User (U) – owner of the file
      * Group (G) – Group owner of the file
      * Others (O) –
    - Permissions
      * Read (R)
      * Write (W)
      * Execute (E)
    - Permissions in Octal
      * Octal – base 8 number system [0…7]
      * 0 🡪 no permissions
      * 1 🡪 Execute
      * 2 🡪 Write
      * 3 🡪 Write & Execute
      * 4 🡪 Read
      * 5 🡪 Read and Execute
      * 6 🡪 Read and Write
      * 7 🡪 Read, Write and Execute
    - Example:
      * -rwx------ 1 user student 48 Sep 20:40 myfile.txt
      * -------- 🡪 0000
      * -rwx------ 🡪 0700
      * -r--r--r-- 🡪 0444
* Processes and Programs (41-43)
  + Program
    - Executable residing on disk in a directory
  + Process
    - Instance of a program currently in execution
    - Programs may spawn multiple processes
    - Defined by PID’s
  + Job Control
    - $ <executable> arg1 arg2 … &
      * Appending command with *&* runs process in background
    - Move bash job between
      * fg --- move background job to foreground
      * bg --- move suspended job to background and resume
  + Standard I/O and Redirection
    - Every process has
      * Standard input (stdin)
      * Standard output (stdout)
      * Error (stderr)

notes2: slides 7-35

* Fundamentals of C
  + Primitives
    - integer (int)
    - floating point number (float)
    - character (char) – single byte that is actually a number
      * 1 byte = 8 bits
    - short integer (short)
    - long integer (long)
    - double-precision floating point number (double)
    - **Types can also be unsigned**
      * No bool type
  + Literals
    - C is case insensitive here
    - ‘char’ vs “string”
  + Type Conversion
    - Implicit
      * **char**, **short** convert to **int** before operation
        + result: **int**
      * **x** (op) **y** = **z** results in **z**  being the highest precision type of **x**, **y**
      * otherwise type **int**
    - Explicit
      * <type> **x** = **y;**
        + **y** is promoted or demoted to the precision of **x**
      * Consequences
        + **y** is larger in size than **x** can store

fractions can be truncated

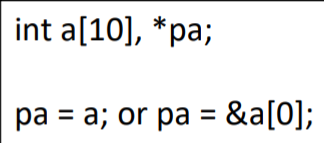
decimals rounded

* + - * + **y** is smaller in size than **x**

not necessarily an increase in accuracy

* + - * + **signed** to **unsigned**

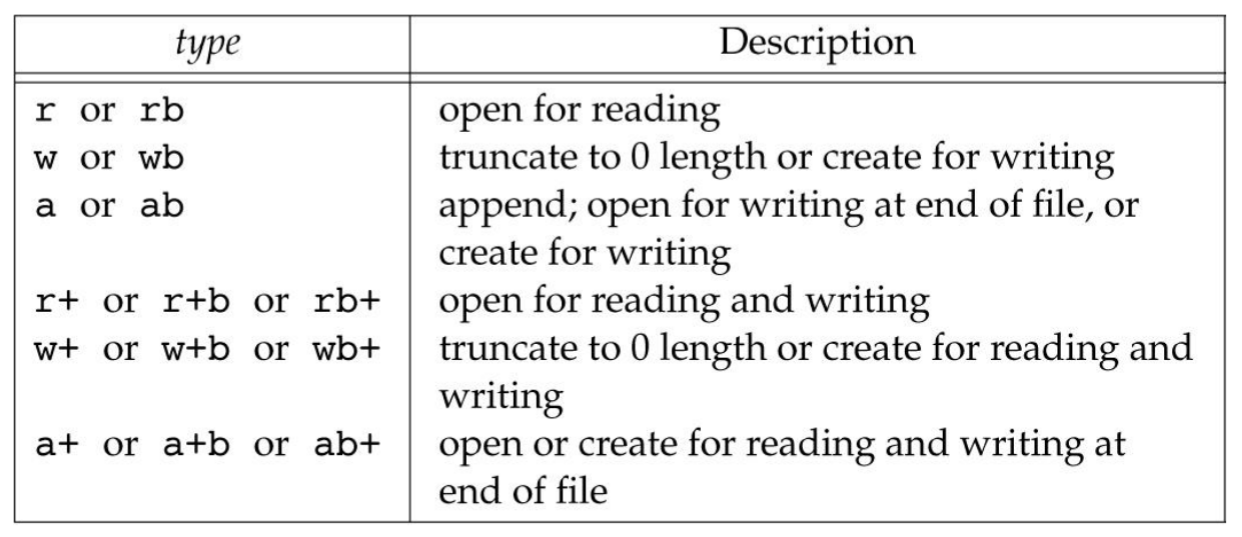
sign may be dropped

* + Constants
    - Statements prefixed with #<statement> = <val> are handled by C pre-processor
      * Pre-processor replaces every instance of <statement> with <val> before compiling
  + Enumeration
    - Organize multiple constants together
    - Values start from 0 and step by 1, unless explicitly specified
  + Functions
    - Visibility
      * Functions can be called anywhere in the program
      * **UNLESS** *static* is prefixed -> restricts visibility to file
  + Pointers
    - Pointer types store an address of memory
      * Typically an **unsigned long** integer
    - Can be **dereferenced** by \*<var> - returns the contents of memory at that address
      * **int** pointer returns an **int**
    - Obtain a reference to <var> by &<var>
  + Arrays
    - Arrays and pointers are intertwined
      * 
      * pa, a, and &a[0] point to the same location
    - A pointer to an array refers to the first element of thearray
      * Iterate over the array by increasing 1 chunk of that type size
      * pa++ or pa—
    - **memcpy** – copy memory area from one portion of memory to another
  + Variable Scope
    - Local - Variables valid within functions
      * Defined within functions
      * Should be defined at the top of the function
      * Allocated in the program’s stack by compiler
    - Global – accessible by the entire program
      * Defined outside functions
      * Allocated in special section of the executable
    - Static – accessible only within the file where the var is defined
      * Defined outside functions
      * Allocated in special section of the executable
        + Programmatically inaccessible to other files. If the memory address is known, it can be accessed
  + Argument Passing
    - C only does call by value
    - Variables passed as arguments become local (copies) for the function in which they were passed
    - Variables can be updated on return
      * i = foo(i)
    - **Pointers** are also passed by value, can update the value of the memory at that address
  + Structures
    - Collection of variables that may be referenced as members of the collective
  + Unions
    - A union takes enough memory to store the largest of its member by only holds one piece of data
    - Accessible like structures
  + Defining Custom Types
    - typedef <type> <new type name>
* Programs in Multiple Files
  + Files compiled individually into object files
    - The **linker** links all object files into one executable
  + Headers
    - Contains declarations of functions, global variables, macros, etc
    - **DON’T DEFINE VARIABLES IN HEADERS**
    - Do define your own types
    - Headers included twice can lead to problems

notes3: slides 3-4, 18-22, 29-37

* File Types (3)
  + Regular files contain data and there is no distinction between text/bin data
  + Directory files contains names/pointers to other files/directories
* File Systems (4)
  + File System (FS) is the collection of methods and data structures that an OS uses to track files on a disk/partition
  + UNIX systems share design elements
    - UNIX File System (UFS)
* File Access Permissions (18-22)
  + Users
    - Have User IDs (UID) and Group IDs (GID)
  + Kernel performs access control according to
    - 1. If super user (UID = 0) 🡪 full access
    - 2. If effective UID of process matches that of file, then user bits of the file are consulted to determine whether access should be granted
    - 3. If effective GID of process matches that of file, then group bits of the file are consulted to determine access
    - 4. The other bits of the file are consulted to determine access
  + Directories
    - **Besides the permissions of a file**
    - All parent directories must have the appropriate execute (E) bit set
      * Also called **search** bit
    - To list contents the read (R) bit must be set
    - To create a file the write (W) bit must be set
  + UID/GID
    - Effective
      * Effective UID/GID are the IDs the process is currently running as
      * Usually the same as the real UID/GID
    - Real
      * Real UID/GID correspond to the user IDs stored in /etc/passwd
      * The account you log in with
    - Saved
      * UID/GIDs saved by **exec** functions
* Files (29-37)
  + Changing Permissions
    - **chmod** or **fchmod**
  + File Links
    - Hard links
      * Act essentially as the same file
        + New link points to the i-node of the original file
        + Actual file is never deleted until all links are gone
      * Cannot hard link across FS (partition)
    - Symbolic Links
      * Links that are pointers to files
        + Link points to the pathname of the file
      * If the file is deleted, symbolic link points to a non-existent file
      * Can point to files in a different FS
  + Deleting a File
    - **Unlink** or **unlinkat**
    - Can delete a file when
      * 1. You own the file
      * 2. You own the directory
      * 3. You are super user
  + Rename a file
    - **rename** or **renameat**
    - Change the name or location of a file
      * File contents are not physically moved on disk

notes4: slides 1-38

* Memory Management
  + Allocate memory - **malloc**
  + De-allocate memory - **free**
  + Allocate a rectangular area and initialize to 0 - **calloc**
  + Resize an area to a size - **realloc**
* Strings
  + Character arrays
    - Arrays of characters terminated by the null char ‘\0’
    - Ex
      * char str[10];
        + string w/ 9 chars + terminating null character
      * char str[] = “hello”;
        + array automatically assigned the string + terminating null character
      * char \*str = “hello”;
        + constant string cannot be updated
    - Array of strings
      * List of lists
  + Conversion functions
    - **ASCII to i**nteger, **l**ong, **l**ong **l**ong, etc.
      * **atoi, atol, atoll, atof, …**
    - Also strtoul, strtoull, …
* Buffered I/O
  + File I/O based on C Streams
    - Streams operate on **FILE** objects and file pointers **FILE \***
      * **Typedef** defined type!
    - Predefined streams
      * 1. Stdin – standard input
      * 2. Stdout – standard output
      * 3. Stderr – standard error
    - Check for EOF’s to discern whether file read error occurred
  + Read/Write
    - Modes
      * 
      * Binaries are not treated differently
    - Can read structures of variables at a time
    - Can jump directly to specified points in a file
      * **fseek**, **ftell**, **rewind**
      * Jump to, get current position and jump to beginning
  + Closing stream
    - Flushes the stream pointed to by a pointer, closes underlying file descriptor
  + Buffering
    - Types
      * **Fully Buffered**
        + Actual write takes place when buffer is full and reads are buffered
        + **stdin, stdout**

when referring to files

line buffered when refer to interactive device (terminal)

* + - * **Line Buffered**
        + Actual write takes place when ‘\n’ is written or when buffer is full and reads are buffered
      * **Unbuffered**
        + I/O takes place immediately
* Error Handling
  + **Errno** is a global integer variable set by libc functions
  + When an error occurs, it is set to a **non-zero** error number
    - Never set to 0 by libc, but it is modifiable
  + Constants corresponding to such numbers are defined in **errno.h**
  + Functions
    - **Perror** – produces an error message including the provided string and a description of the last error
  + Terminating a process
    - **exit(**int status**)** after **fatal** error
    - all open streams are flushed and closed
    - **EXIT\_SUCCESS** or **EXIT\_FAILURE**

notes5: slides 1-14, 19-21

* Low Level File I/O
  + Buffering
    - Reduce the number of inefficient I/O operations
  + File Descriptors
    - Non-negative integer that corresponds to a file opened by the process
    - Components are abstracted as files in unix
  + Opening
    - **open**
      * system call (POSIX standard)
      * returns file desciptor
    - **fopen**
      * c-std lib
      * may be faster due to buffering
      * portable
      * returns file pointer
    - **openat**
      * Open at relative paths under the directory corresponding to descriptor **dirfd**
      * Useful in multithreaded programs
  + Seeking
    - Actual data files can usually be seeked, but not all descriptors are seekable
* More on File Descriptors
  + When opening file, it is assigned the smallest descriptor
    - File descriptors locate structures that hold file metadata
  + Files can be opened multiple times by different processes
  + Each descriptor maintains its own flags

notes6: slides 1-40

* Linking (1-31)
  + Static Linking
    - Programs are translated (compiled separately) then linked using a compiler driver
    - Results in a fully linked executable object file (contains code and data for all functions)
  + Why?
    - Modularity
      * Program can be written as a collection of smaller source files
      * Build libraries of common functions
      * Avoid nasty errories
    - Efficiency
      * Time: Separate compilation means that the whole project does not need to be recompiled after one source file is changed
      * Space: Libraries can be aggregated into a single file
    - When?
      * **Compile time** (when compiled)
      * **Load time** (when loaded in memory)
      * **Run time** (when executing)
  + What do linkers do?
    - Step 1: Symbol resolution
      * Programs reference symbols (global vars and funcs)
      * Symbol definitions are stored in the object file (by the assembler) in **symbol table**
        + An array of **structs**
        + Includes name, size and location of that symbol
      * Each symbol reference is as associated with exactly one symbol definition
    - Step 2: Relocation
      * Merge separate code and data sections into single sections
      * Relocate symbols from relative locations in **.o** files to their final locations in the executable
      * Update all references to symbols to reflect new positions
  + Object Files (Modules)
    - 1. Relocatable object file (**.o**)
      * Contains code and data that can be combined with other **.o**’s to form an executable object file
      * **.c** 🡪 **.o** files are one to one
    - 2. Executable object file (**.out**)
      * Contains code and data in a form that can be copied directly into memory and executed
    - 3. Shared object file (**.so**)
      * Special type of relocatable object file that can be loaded into memory and linked dynamically (at load time or run-time)
      * **Dynamic Link Libraries** (DLL’s)
  + Executable and Linkable Format (ELF)
    - Standard binary format for object files
  + Linker Symbols
    - Global
      * Symbols defined by a module that can be referenced by other modules
      * **Non-static variables** and **functions**
    - External
      * Global symbols reference by a module by defined by some other module
    - Local
      * Symbols defined and referenced exclusively by the individual module
      * Local linker symbols **are not local program variables**
  + Linker Symbol Rules
    - 1. Multiple strong symbols are not allowed
      * Can only define an item once
      * Cannot link two files that both declare global function f()
    - 2. Given a strong symbol a multiple weak symbols, choose the strong symbol
      * References to ‘weak’ symbols resolve to strong symbol
        + Weak symbol: *int a;*
        + Strong symbol: *int a = 5;* or *int p1() { … }*
    - 3. If there are multiple weak symbols, pick an arbitrary one
  + Static Libraries (**.a** archive files)
    - Old fashioned packaging of commonly used functions
    - Concatenate **.o**’s into a single file with an index (**archive**)
    - Enhance linker so that it tries to resolve unresolved external references by looking for symbols in one or more archives
    - If archive member resolves reference, link archive to executable
    - **Problems**
      * Command line order matters, include libraries at the end of the command line
      * Duplication in stored executables (every function needs libc)
      * Duplication in running executables
      * Minor bug fixes require explicit relinking
  + Shared Libraries (**.so** or **DLL**s)
    - Object files containing code/data are loaded and linked at load-time or run-time
    - Shared libraries can be shared by multiple processes
    - **Load time**
      * Create partially linked executable file
      * Then, in **memory**, the dynamic linker creates the fully linked executable
    - **Run time**
      * Load (and unload) the shared library in the code that does the loading (and unloading) while the code is being executed
* Make files (32-40)
  + Automatically re-builds an executable after source files have changes
    - Specifically compares the last-write-time of target file (executable) vs. that of dependencies (source files)
    - If any is newer, or target doesn’t exist, executes the command associated with the target
    - Requires that dependencies be declared correctly

notes7: slides 1-41

* Processes (1-12)
  + Instance of a running program
    - Life begins when **exec**uted
      * Usually ends up calling its **main()** function
    - Life ends when main **returns**
      * Or if a function or system call terminates it
  + Exit functions
    - **return <nr>** is equivalent to **exit <nr>**
      * Calls all handlers installed with **atexit()** and **on\_exit()**
      * Closes I/O streams
      * Calls **\_exit** or **\_Exit**
    - **\_exit** or **\_Exit**
      * Tells kernel to immediately close all files and terminate process
  + Exit Handlers
    - Functions or processes to run/execute on current process exit
  + Sections of object/archive files
    - **Text**
      * Actual machine instructions that the CPU will execute
    - **Data**
      * All initialized variables (declarations)
    - **BSS**
      * Uninitialized data such as arrays that values or pointers have not been set
* Process Control (13-41)
  + PIDs
    - ID assigned by kernel to process
    - ID of terminated process can be re-used
  + Creating processes
    - **fork()**
      * Create child process
        + Child is a copy of the parent
        + Inherits most resources/properties
      * Both parent and child resume execution in the same instruction following fork
      * 0 🡪 child process PID
      * Child PID 🡪 parent process PID
        + Parent has no other way of retrieving child PID
      * Parent and child processes both retain their own memory copies/allocations/standard I/O buffers
      * **Differences**
        + Return values
        + PIDs
        + Different parent PIDs
        + Child’s time values are set to 0
        + File locks set by parent are **not inherited by child**
        + Alarms and signals are cleared for child
    - File Sharing
      * Child process inherits all file descriptors of parent
  + Zombies
    - Processes do not completely disappear until their parent waits for them
      * They become zombies until that happens
      * Ensures that parent can retrieve their exit code
  + **exec** functions
    - Replaces the current running process with a new process
    - Run a c program by using another c program
  + Least-privilege model
    - Always use the least privileges needed for a job
      * Mitigate damage when shit happens
    - Processes running as root drops its privileges
  + Process accounting
    - Produce info about create and terminated processes
      * Children of current process

notes8: slides 1-22

* Exceptional Control Flow (1-5)
  + Processors only do one thing
    - CPU reads and executes a sequence of instructions one at a time
    - Use **control flow** to react to changes in **system state**
  + ECF
    - Low level
      * 1. Exceptions
        + Change in CF in response to **system state**
    - High level
      * 2. Process context switch
        + OS
      * 3. Signal
        + OS
      * 4. Nonlocal jumps
        + C runtime lib
* Exceptions (5-22)
  + Transfer of control to the OS ***kernel*** in response to some event
    - **Kernel** is the memory-resident part of OS
  + **Asynchronous**
    - (interrupts) Caused by events external to the processor
      * Set’s *interrupt pin*
    - Examples
      * Timer interrupt
        + External time chip triggers interrupt
      * I/O interrupt
        + Hitting ctrl-c
  + **Synchronous** 
    - Caused by events that occur because of executing an instruction
      * **Traps**
        + Intentional **system calls**, breakpoints or special instructions
        + Returns control to next instruction
      * **Faults**
        + Unintentional but possibly recoverable
      * **Aborts**
        + Unintentional un unrecoverable
        + Illegal instruction, parity error, machine check
        + Aborts current program
  + System call
    - All have unique ID number
    - Actions of the kernel
  + Multiprocessing
    - The **Illusion**
      * Computers running my processes simultaneously
      * Applications and background tasks
    - The (**traditional**) reality
      * Single processor executes multiple processes concurrently
        + Process executions interleaved (multitasking)

Alternate between processes/instructions

* + - * + Address space managed by virtual memory
        + Register values for nonexecuting processes saved in memory
      * Save current register in memory
        + Schedule next process for execution
        + Load saved registers and switch address space (context switch)
    - The (**modern**) reality
      * Multicore processors
        + Multiple CPUs per single chip
        + Share main memory and some of the caches
        + Each can execute a separate process

Scheduling of processes onto cores done by kernel

* + Concurrent Processes
    - Each process is a logical control flow
    - **Concurrent**
      * The flows of two processes overlap in time
      * Otherwise sequential
    - Control flows for *concurrent* processes are physically disjoint in time
    - However, we can think of concurrent processes as running in parallel with one another
  + Context Switching
    - Processes are managed by the Kernel
      * The kernel is not a separate process, but instead runs as part of some existing process
    - Control flow passes from one process to another via **context switch**
      * The kernel redirecting which process is executing

notes9: slides 1-18

* Process Relationships (1-8)
  + Shell Programs
    - Shell is an application program that runs on behalf of the user
    - Simple shell waits for and reaps foreground jobs
      * What happens to background jobs? 🡪 ECF
    - Exceptional Control Flow
      * Kernel interrupt regular processing to alert when a background process completes
      * Alert mechanism is called a **signal**
* Signals (8-18)
  + **Signal**
    - Small message that notifies a process that an event of some type occurred
  + Sending signals
    - Kernel *sends* signal to a *destination* *process* by updating some state in the context of the destination process
    - Why?
      * Kernel detected a system event
      * Another process invoked a ***kill***system call to explicitly request the kernel to send a signal to the destination process
  + Receiving signals
    - A destination process **receives** a signal when it is forced by the kernel to react in some way to the delivery of the signal
    - Possible reactions
      * **Ignore**
      * **Terminate**
      * **Catch**
        + Catch the signal by executing a user-level function called **signal handler**
  + Pending and Blocked signals
    - **Pending**
      * If a signal is sent but not yet received
      * Can be at most one pending signal of any particular type
        + Signals are **not queued**
    - **Block**
      * Blocked signals can be delivered but will not be received until the signal is unblocked
      * Uses bit mapping to decide which signals to send
        + Pending & ~blocked
  + Process Groups
    - Each process belongs to exactly one process group
      * PGID
    - Can send signals to process or process group